C# Design Patterns: Memento

APPLYING THE MEMENTO PATTERN

Steve Smith
FORCE MULTIPLIER FOR DEV TEAMS
@ardalis | ardalis.com | weeklydevtips.com

Objectives

What is the memento pattern?

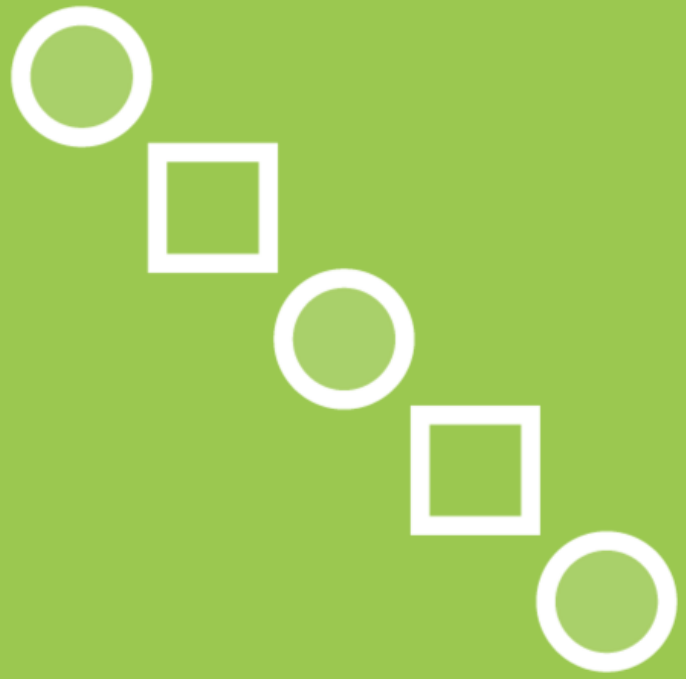
What problems does memento solve?

What is the structure of the memento pattern?

How to apply the pattern in real code?

How to recognize related patterns?

A memento holds an object's internal state so the object can be restored to this state later.



Memento is a *behavioral* design pattern.

Examples of Operations

Saving state in games

Supporting undo in a drawing application

Rolling back a distributed transaction

What Problem Does Memento Solve?

Memento Applicability

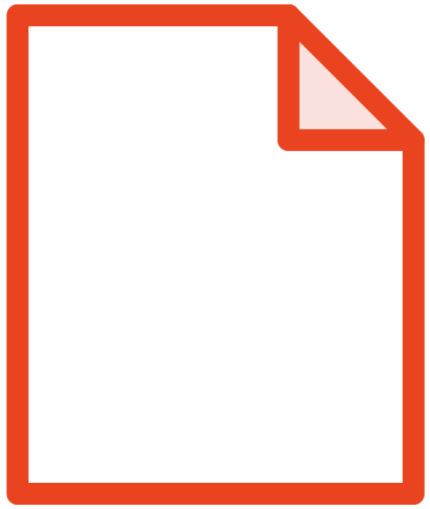
Need to “roll back” one or more objects to a previous state

Adding undo to existing objects would violate Single Responsibility Principle

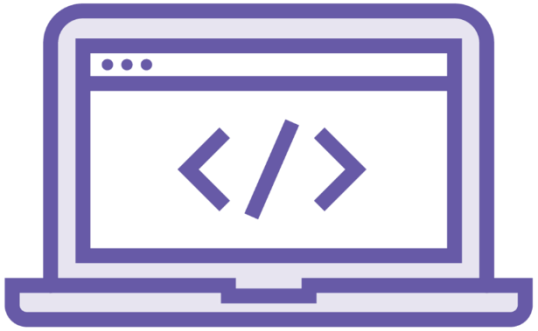
Providing full, direct access to objects’ internal state breaks encapsulation

What Is the Structure of the Memento Pattern?

Memento Collaborators



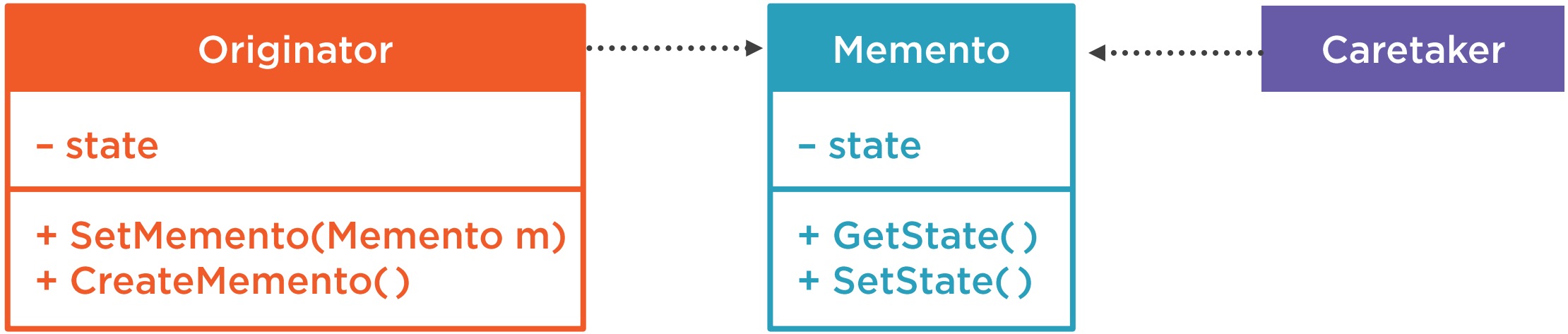
Originator



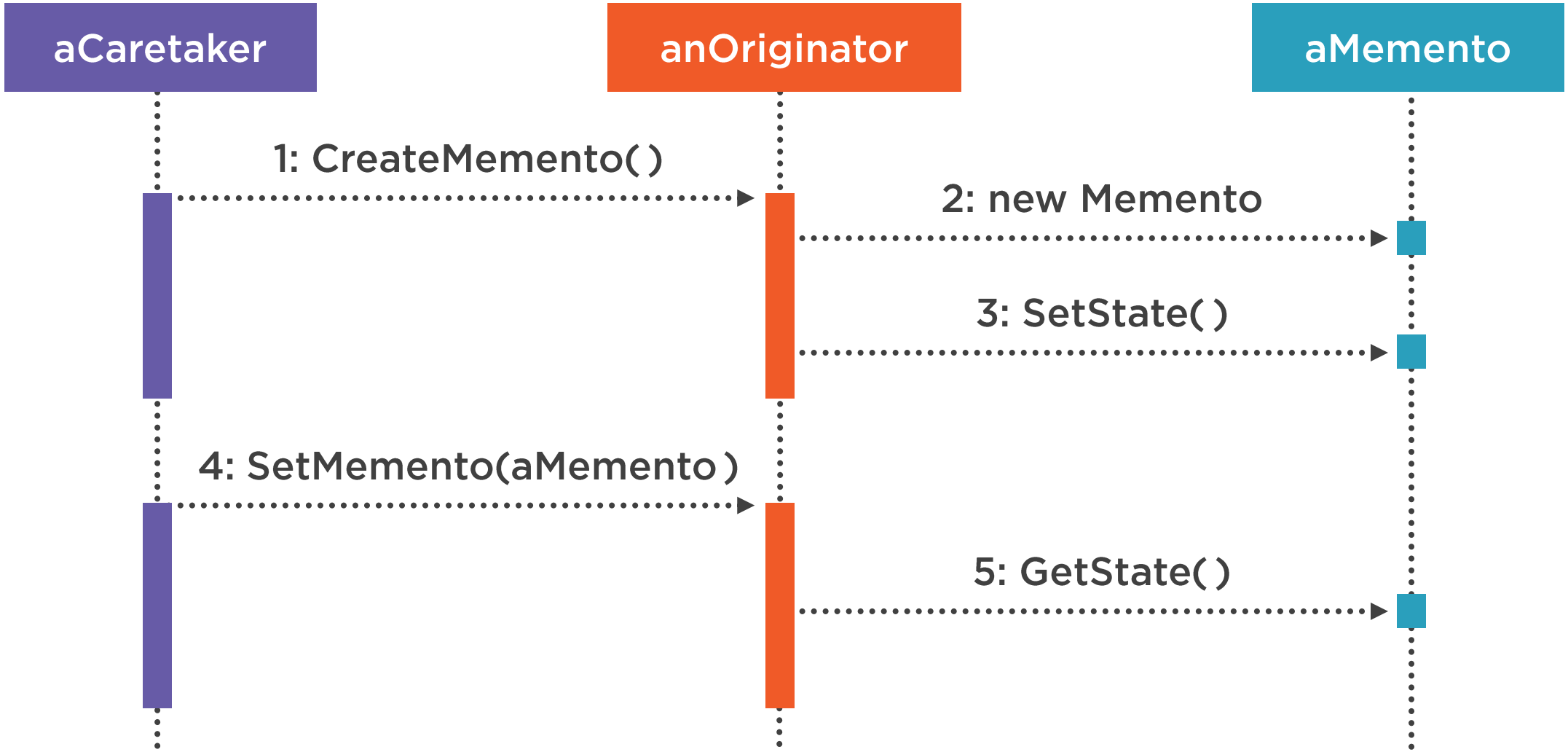
Caretaker

Memento

Memento


Originator
– state
+ SetMemento(Memento m)
+ CreateMemento( )
Memento
– state
+ GetState( )
+ SetState( )
Caretaker

Memento


aCaretaker
anOriginator
aMemento
1: CreateMemento( )
2: new Memento
3: SetState( )
4: SetMemento(aMemento)
5: GetState( )

Working with Memento

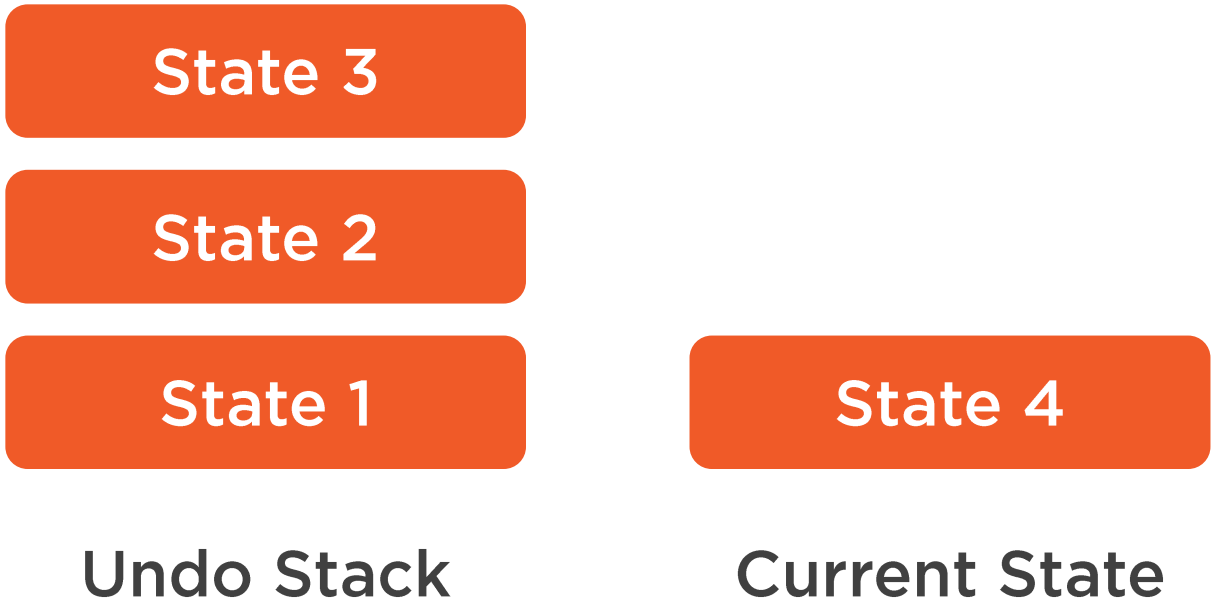
Memento itself should be very simple

Originator must support methods to create/restore mementos

Caretaker is responsible for managing previous states

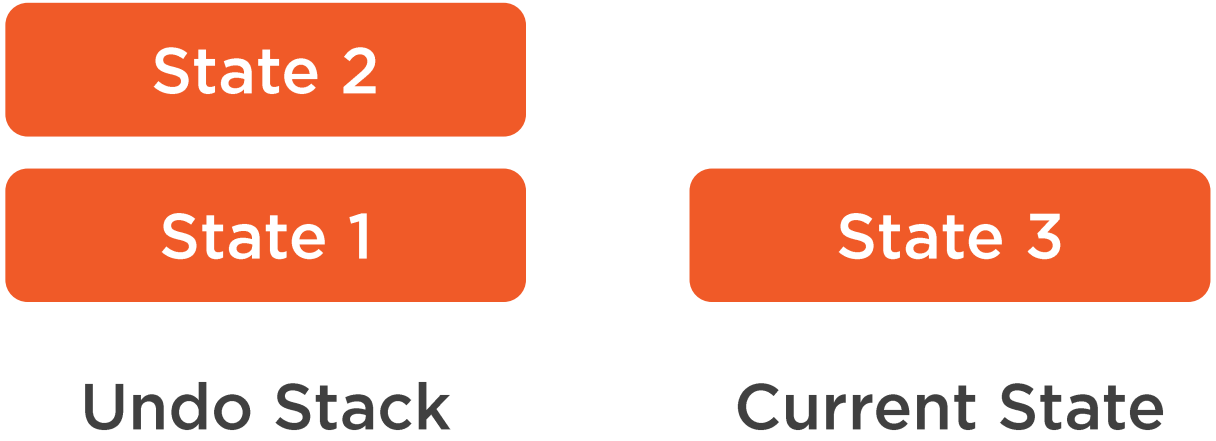
Avoid giving caretaker direct access to internal memento/originator state

Undo Stack

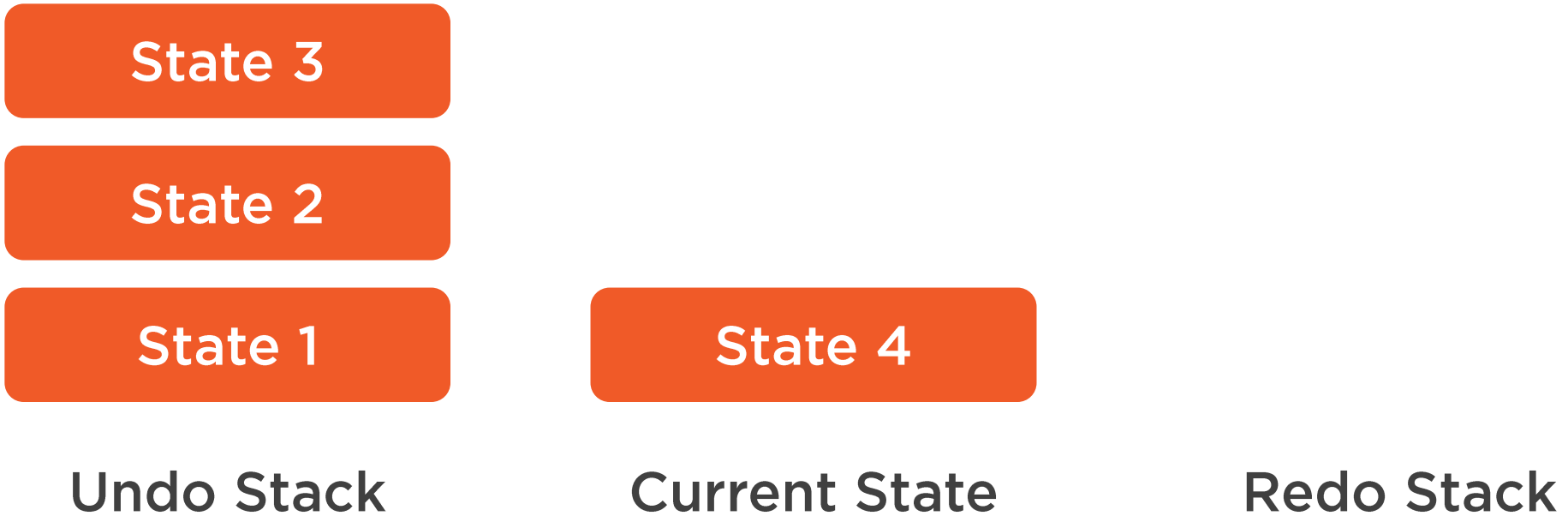


Undo

Undo Stack



Undo/Redo Stack Drawing



Undo

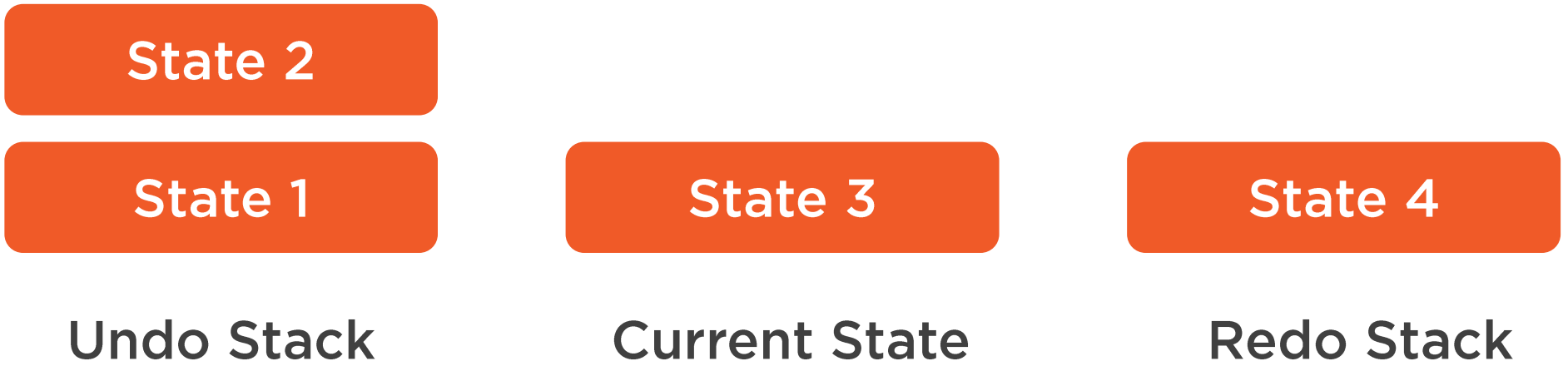
Undo/Redo Stack Drawing

Undo

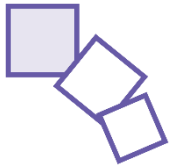
Undo/Redo Stack Drawing

Redo

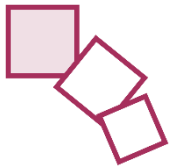
Undo/Redo Stack Drawing



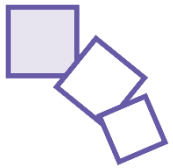
Implementing Undo/Redo



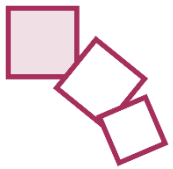
Store states (mementos) on an undo stack



After each action, add new memento to undo stack



On undo, pop previous memento from undo stack; add to redo stack



On redo, pop from redo stack; add to undo stack

Mementos should be immutable value objects with state, but no behavior.

How Do We Apply Memento to Existing Code?

Steps to Apply Memento

Follow refactoring fundamentals

Define a Memento type

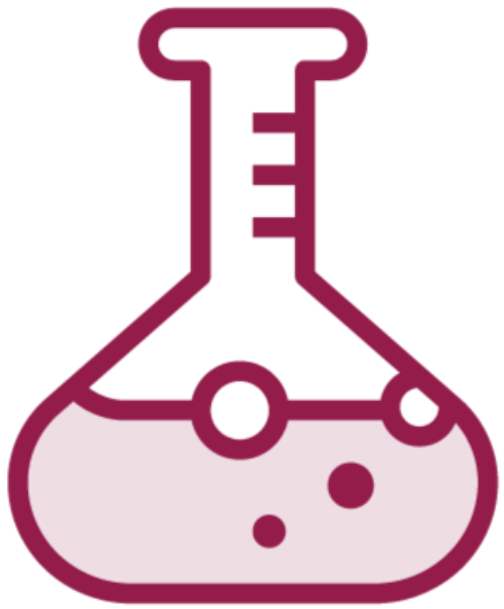
Add save and restore methods to Originator

Manage stored states in caretaker

Demo

Applying the Memento pattern in a simple console game

Analysis



Only Caretaker, not Originator, has to track state

Memento may not be appropriate if state is quite large

Can be difficult to encapsulate memento state so only Originator can access it

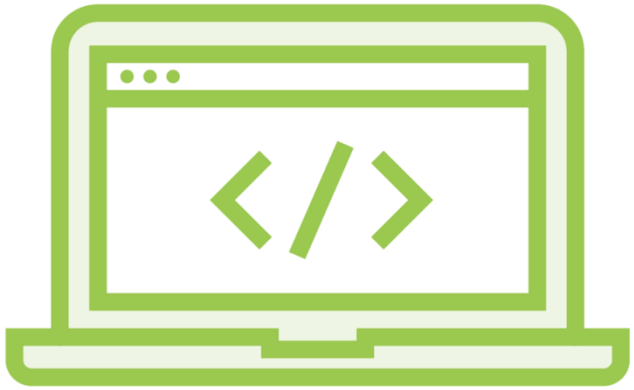
Alternate Approach: Reverse Operations

Works well with Command pattern

Store operations performed

Support undo by applying reverse operation

Only works for operations with consistent reverse behavior



Reverse Operation Example: Calculator

Start with: 30

Add(10)

Multiply(2)

Undo => Divide(2)

Undo => Subtract(10)

End with: 30

Reverse Operation Example: Calculator

Start with: 10

Subtract(20)

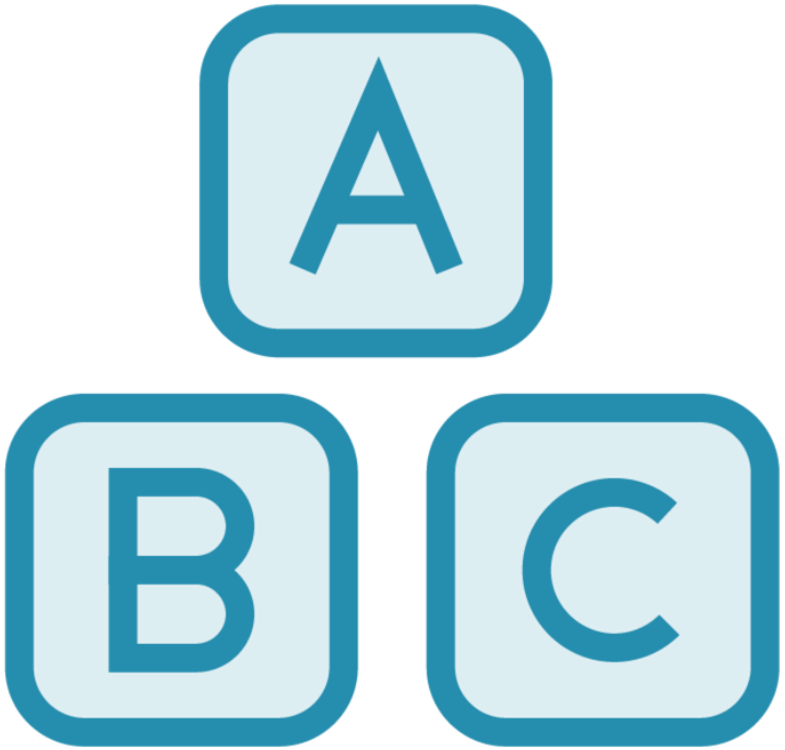
Square()

Undo => Sqrt()

Undo => Add(20)

End with: 30

Reverse Operation: Translation



"Pluralsight delivers quality training to developers worldwide."

"Pluralsight bietet Entwicklern weltweit qualitativ hochwertige Schulungen."

"Pluralsight предлагает высококачественное обучение разработчиков по всему миру."

"Pluralsight offers high quality developer training courses worldwide."

Alternate Approach: Storing Diffs

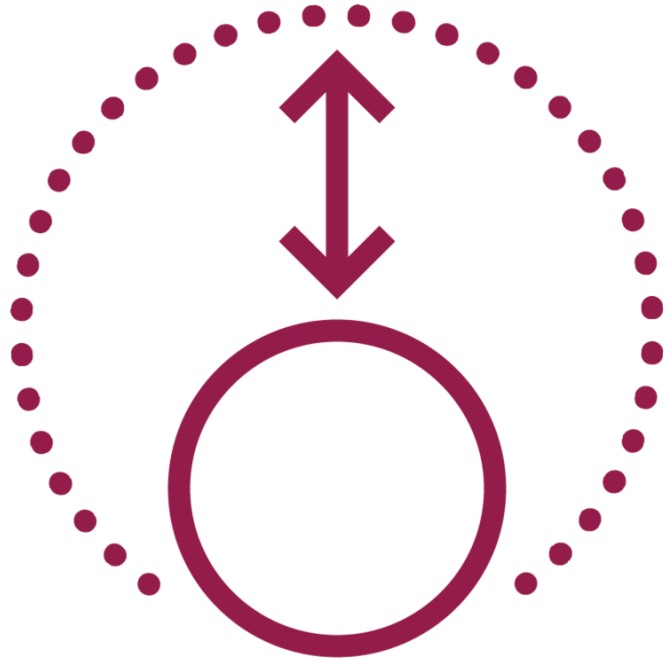
Instead of storing full state, just store differences

This is how version control systems like Git work

Requires less storage to save many states

May require more resources to create diffs

May require more resources to restore a state that involves many diffs



Related Design Patterns

Command

Provide reversal commands
to undo operations

Iterator

Each iteration can store
its state using a Memento

Key Takeaways

Memento Design Pattern

- Behavioral Pattern
- Stores state of an object (Originator)
- Removes state management from Originator's responsibilities

Common uses:

- Game save points
- Undo support
- Undo/Redo support

Key Takeaways

Key Principles:

- Single Responsibility Principle
- Encapsulation

Refactoring steps

Alternate Approaches

Related patterns

- Command
- Iterator